DESIGN ISSUES FOR Ada PROGRAM SUPPORT ENVIRONMENTS

A CATALOGUE OF ISSUES

SCIENCE APPLICATIONS, INC.

DESIGN ISSUES FOR Ada PROGRAM SUPPORT ENVIRONMENTS

A CATALOGUE OF ISSUES

SAI-81-289-WA

October 1980

Dr. David A. Fisher

This report was supported by the Defense Advanced Research Projects Agency under DARPA Order No. 3456, Contract No. MDA903-80-C-0188, monitored by the Defense Supply Service, Washington, D.C. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Defense Advanced Research Projects Agency of the United States Government.

ATLANTA • ANN ARBOR • BOSTON • CHICAGO • CLEVELAND • DENVER • HUNTSVILLE • LA JOLLA
LITTLE ROCK • LOS ANGELES • SAN FRANCISCO • SANTA BARBARA • TUCSON • WASHINGTON

SCIENCE APPLICATIONS, INC.
1710 Goodridge Drive, 12th Floor, McLean, Va. 22102

ACKNOWLEDGEMENTS

The author would like to thank John Buxton, Tom Cheatham, Steve Crocker, Larry Druffel, Steve Squires, Tim Standish, and Bill Wulf for their thorough and insightful reviews of a draft version of this paper and for the many helpful comments and suggestions they have provided.

DESIGN ISSUES

FOR

Ada PROGRAM SUPPORT ENVIRONMENTS

A CATALOGUE OF ISSUES

TABLE OF CONTENTS

I. Introduction

This paper catalogues many of the design decisions necessary to the design of an Ada program support environment (APSE). The emphasis is on those design decisions which affect the APSE users and tool builders. It is hoped that early identification and resolution of these design decisions will maximize the simplicity, usefulness, and ease of use for any APSE while permitting tool building efforts to begin in parallel with implementation of the kernel APSE. The catalogue includes both design decisions and conventions on the form, function and performance of the various internal and external interfaces, but does not address other design decisions critical to the implementation and internal representations of the kernel APSE.

[illegible] host computer programming languages, compilers,

The APSE and kernel APSE requirements as given in the Stoneman [1] and its supporting documents [2,3,4,5] are assumed to be valid. Particularly important is that the Ada programming language will be an integral part of any APSE so that the Steelman requirements [6] will be satisfied by any APSE and the kernel APSE will provide direct access to all Ada features in the host environment without providing duplicative alternative APSE concepts to satisfy requirements (e.g., multi-tasking, synchronization) that have been dealt with adequately in the Ada design [7]. Although neither new requirements nor rationale for existing requirements are given in this paper, Stoneman and Steelman requirements will sometimes be paraphrased to provide context. It is assumed throughout this paper that the reader is familiar with the vocabulary and concepts of the Stoneman and of Section 13 of Steelman.

It should be remembered that Ada and APSEs are intended primarily for the design, development, test, maintenance, and evolution of military embedded computer applications and other similar systems. Such systems are characterized as large (50 to 100 thousand source lines and larger), long lived (15 to 20 years or more) and continuously changing (with annual changes sometimes of the same magnitude as the total system). Such systems are also characterized by real time constraints, the need to interface and control nonstandard input-output devices, the need for concurrent processing and control, and requirements for fail soft and fail safe execution.

The catalogue of design decisions is given in an outline form which attempts to group related issues. The design of the kernel APSE should specify the specific design decision or the proposed convention for each item of the outline. Wherever possible the design should take form of an Ada package specification giving the calling format for the operations and an English language description of their function.

II. Ada Program Forms

A variety of logical representations is needed for Ada programs to support compilers, system design, derivation and implementation tools.

A. General Form. A general purpose from which encompasses the needs of all compilers and tools is needed. It must be open-ended because all the specific needs cannot be anticipated at the time of its design. It should provide operations for accessing and modifying the information content of objects, for annotating objects with attributes, and for traversing networks formed by relationships between objects.

B. Compiler and Tool Intermediate Forms. A variety of intermediate representations for Ada programs is needed in any compiler as a function of the phase within the compilation process, the compilation technique being used, the degree of optimization being applied, and the target machine characteristics. Never-the-less, several specific intermediate forms can be identified for any compiler and their common attributes specified. For each the design should give the form as a specialization of the general form (see II. A above). It should identify the specific logical forms, attributes and interpretations for each representation used at the interfaces between the parts of the accompanying compilèr, and should distinquish between attributes common to all compilers and those which are unique to the accompanying compiler.

1. Parser Output. The intermediate form for parsed Ada programs to be used by language editors and pretty printers and as input to the target machine independent phases of compilation. Data structure, operations, and interpretation of any accompanying symbol or other tables should be included.

2. Semantically Enhanced Form. The intermediate form in which the compile time Ada semantic checking has been completed and other semantic information about the source program may be encoded in an easily accessable form. Typically their general form would be used in various phases of analysis, transformation, and target independent optimization of programs.

3. Code Generation Form. This intermediate form provides the interface between the "front end" and the code generation phase of compilers. The form should be target

machine independent, but might have attributes dependent on the architectural group of the target (e.g., general register vs. stack) or on the degree of machine dependent optimization to be applied. It can be more efficient than the Semantically Enhanced Form because much of the earlier analyses information is not needed by the code generator or because the specific uses of the information within code generators is well understood. It might be specialized for some use such as debugging of gathering statistics.

C. Ada Program Input-Output Forms. There are several forms required for the exchange of Ada programs between systems. Unlike the forms of II. A. and II. B above which can be described solely in terms of their interface characteristics, the Input-Output forms must have a host system independent physical representation as well.

1. Pretty Print Form. There should be a specified format for indentations, etc. for the exchange of source programs, publication of algorithms in Ada, and supported by Ada language editors.

2. Inter-Host Transfer Form. There should be a specified format for the exchange of programs in intermediate language form between host systems. This can be done by providing a one-one translation between intermediate language and character string data (a la TCOL), by providing a machine independent form for intermachine communication of any Ada data structure including intermediate structures (see also IV. G.4.), or both. Note that there is not a semantic difference between inter- and intra- host transfers. In both cases there are lots of attributes generated by various tools, but not needed or supported by all tools. The inter-host form must provide a mechanism for efficient transfering of the needed attributes over relatively narrow bandwidth channels.

3. Code Representation Form. There must be a form for transfer of target code from host to target machines. Although such forms must be target machine dependent, they should be host machine independent, there should be only one per target instruction set architecture (ISA), and there should be

common conventions for relocation, parameterization, error checking and diagnostics at load time. Common characteristics for all code forms and specific representations and conventions for each code generator of the accompaning compiler should be provided with the design.

III. Command Language and Executive Functions

The kernel APSE must provide common interfaces and conventions among host users and host system programs, a command language for directing and scheduling host system functions, conventions to ensure commonality in the use of tools, and a comprehensive set of host system utilities. Most of the issues below will be equally important in APSE and tool design. The idea is to satisfy the critical requirement for consistency throughout the APSE by establishing common conventions and accessable implementations of those conventions at the level of the kernel APSE.

A. External Interface Conventions. Standard interface processes are required as intermediaries between the external user of the host system and any host system program (whether a host user program, software tool, system utility, or the command language itself). The interface provides a common set of conventions to the user regardless of the subsystem being used and provide a common interface to system functions regardless of the users terminal characteristics. Separate interface routines are required for the three major interface device categories (i.e., batch, linear interactive, and high bandwidth graphic interactive) because their bandwidth and display characteristics dictate major differences in their interface granularity (i.e., in the size and frequency of interactions). The kernel or minimal APSE design should show what facilities and conventions are provided for such characteristics as the following:

1. Character, word and line input error correction for interactive input. These might take the form of commands entered through an editor or interface routine consistent with the kind of user interface.

2. Standard means (or control keys) for terminating commands.

3. Standard command, control key or whatever for ending (i.e., normal termination) of any subsystem.

4. Standard form, conventions, kernel facilities, and controls for invoking, interpreting, and responding to subsystem help facilities and diagnostics.

5. Standard conventions and facilities for undoing and redoing commands.

6. Conventions and controls for user requested command completion and typing extension (i.e., for system extension of user typing of commands or arguments in unambiguous contexts and default situations.

7. Conventions, controls and meaning of returning to system command language level.

8. Conventions and controls for user generated asynchronous interrupts for command or program interruption and output termination.

9. Facilities for user alteration of any of the above conventions or control key assignments as a function of user preference or terminal characteristics as specified in a user profile.

10. Conventions on how each of the above are communicated to or interact with utilities, tools, or user programs within the host system. The internal interfaces should be described in terms of requirements and conventions for the host system Ada programs implementing the utilities, tools, or user programs (e.g., external controls might appear as end of record on read, a raised exception, an entry call or accept, or as a global Boolean variable).

B. Command Language Syntax and Interpretation. The command language is used by the external user to invoke tools and host programs, control their scheduling, query the system, parameterize system functions, interact with running programs and tools, request immediate execution, and conduct interactive debugging. Thus, the command language as discussed here combines the interactive interface functions of interpretive languages and debugging systems with the job control functions of tradtional batch systems. The goals of simplicity, ease of learning and understanding, and minimization of concepts all suggest that the command language for an APSE should be as similar to Ada as possible. The differences in the purpose and uses of command and progamming languages however impose a number of considerations that must be dealt with in the command language design.

1. Manipulatable Objects. Because the command language is used to create, modify, test and query Ada programs and their activations, command language must not only have all the Ada types but must also provide a definition for programs, packages, variables, declarations, types and other Ada compile-time, run-time entities as types defined in Ada, and probably others.

2. Composition. Because the protocols of interaction, let alone their function, cannot be known at the time of APSE design for all systems commands, development and maintenance tools, testing and debugging systems, and interactive host applications that eventually will be needed, the command language must provide a complete facility for defining, composing (i.e. functional composition), and retaining (i.e. as in a library) operations. Composition is a primary characteristic of general purpose programming languages (including Ada), but not of many job control languages. These considerations coupled with the desire for simplicity and minimization of concepts in the APSE impose a requirement for an Ada based command language providing access to the full composition facilities of Ada.

3. Syntactic Issues. Although the full generality of the Ada language (and its syntax) must be available to the command language user, typical job control language usage has been one operation at a time with arguments that are literals rather than expressions or variables. The command language might provide special syntax for this kind of use or show that the facilities for command completion and typing extension (see III. A.6) satisfy the need. The latter approach is particulary desirable because it will not require any syntactic extensions.

4. Semantic Differences. Any differences in interpretation of declarations, statements or expressions in the command language from those in Ada programs should be detailed in the design. These might include immediate elaboration of declarations or execution of expressions prior to entry or compilation of subsequent statements, alternative processing for exceptions (e.g., report to user at terminal rather than terminate task or program), and access to variables external to programs.

5. Context. The design must clearly define how the user specifies, alters, and controls the context of execution of command language expressions and statements (i.e., command language scope and visibility rules).

6. Display. The user should be able to obtain automatic display of results of command language expressions executed from the terminal. The design should define when and how this is accomplished. Note that this is trivial in expression languages.

C. Host Process Control and Scheduling. Among the most mportant command language facilities are those for control and cheduling of user host processes. Any given user may have everal activities going on simultanously, may need to pass data etween them, may need to suspend one activity to accomplish an uxiliary activity (e.g., send message to get a routine to edit, r compile and execute to obtain an argument to the current ctivity), and to change the process to which the users terminal s attached.

1. Processes. If there is to be any difference between the host command language process mechanism and the Ada facilities for task activation, rendevorus, delay, and selection, they should be detailed in the design and strong justification provided.

2. Terminal Connection. The command language should have operations for specifying and changing which of the user°s processes are interacting with the terminal. The design should specify the effect of terminal input or output requests from processes not currently interacting with the terminal, whether the set of processes for a user are organized in a forest or hierarchical structure, and whether terminal and process associations are made explicitly or implicitly.

3. Scheduling. The scheduling discipline to be used among independent processes of a given user and among processes of different users of the same host system is beyond the scope of the Ada language semantics and thus cannot be adopted directly. The kernel APSE design should specify a standard algorithm or provide a mechanism for user control of such scheduling.

4. Query. How and to what extent can processes (and users) interrogate the status of the system, the associated user terminal, the user°s accounting information, the terminal characteristics, etc.? The design should specify the operations for query of system variables and status, to what variables they apply, and what protection is provided against improper use. Note that all of this can be done in Ada.

IV. Host System Database

The database is the central feature of an APSE, acts as the repository for all information of a project throughout the project life cycle, and provides the central mechanism for operation of APSE tools. The kernel APSE design must provide a complete logical description of the database facilities including the set of operations (actually Ada subprogram specifications) to create, modify, restructure, access, assign, or otherwise manipulate databases.

A. Objects. Each distinguishable component of the database including files will be called an "object". The kernel APSE must provide several specific properties for objects.

1. Name. Each object must have a unique internal name that can be used by any program or tool, can be passed as arguments and stored in variables (i.e., objects act like and might be implemented as Ada objects of access types).

2. Type. Any entity to which a user, tool, or system designer might want to refer must be representable as an APSE object. Thus Ada programs, subprograms, packages, expression, varables; APSE tools, symbol tables, and program representations; any Ada data type; types, declarations, identifiers, and control structure; and configurations, versions, groups, and partition are all examples of APSE objects.

3. Relations or Annotations. The kernel APSE must provide operations to establish, modify, and interrogate relations among objects.

B. Directory. The kernel APSE or at least the minimal APSE must provide a directory system that associates (external) symbolic names with objects of the database. Symbolic names shall be constructed as a sequence of identifiers. Although it will be possible to uniquely identify an object by its full name sequence, it should be possible to provide a partial name and to obtain disambiguation on the context (e.g., the current user), the intended usage (e.g., Ada overload resolution on type, or use restricted to a particular partition), and defaults (e.g., the current version). The design should detail the directory system and operations.

C. Versions. A group of objects may exist within the database as different versions of the same "abstract object". The design must allow for the use of a single (internal) name for the abstract object with implicit selection of the relevant version based on the use requirements, type or partitions. An abstract Ada program object, for example, might have versions for each level of its derivation from an abstract program description, versions for various phases of the compilation process, and code versions for several target machines, but an attempt to print the abstract object as an Ada program would select the pretty print symbolic version.

D. Configurations. The APSE is intended for use in large systems involving many people concurrently and over long periods of time. The programs and software systems to be developed and maintained are not simple objects, but involve large collections of objects in many versions undergoing change from many quarters. The kernel or minimal APSE must provide a mechanism for establishing stable configurations from designated fixed versions of the component objects, for creating new local configurations by incremental modification, and for ensuring that no component object of an accessible configuration can be deleted or modified.

E. Partitions. Partitions are a mechanism for dividing parts of the database into a variety of collections for various purposes some of which are listed below. Partitions are a generalization of the Ada type mechanism (and thus used for version disambiguation in APSEs). Unlike types however, objects can be dynamically added or removed from partitions and partitions need not be disjoint. The kernel APSE should provide a general mechanism for user definition and modification of partitions. There may be a need for hierachical indices to the components of a partition. For each of the partitions listed below, the kernel APSE design should include a complete definition of the facility and its operations or should show that such a facility can be built by an APSE implementor from primitives provided in the kernel APSE.

1. Accounting. It may be necessary to partition the objects of an APSE according to the account to which the associated storage and execution costs are to be charged. Minimally, the kernel APSE must provide a mechanism for determining the amounts of storage space and execution cycles associated with a given accounting partition.

2. Protection. It must be possible to protect one (logical) "user" of an APSE from another by partitioning the objects and files of the database into collections in which the user controls which objects of his partition can be read, altered or executed by another user.

3. Authentication. An APSE database can be partitioned according to what objects and files are accessable to a given physical user (i.e, person). Typically a user wears several hats and therefore has access to several protection partitions. Similarly, protection partitions are often associated with multiperson projects and therefore overlap several authentication partitions. The kernel APSE should provide a mechanism for user definition of authentication routines and some primitive mechanism such as passwords in the minimal APSE.

4. Storage. The storage media (e.g., main memory, on line peripheral memory, and archival storage) is another important partitioning of the database. Users should be able to specify, query, and control the storage partition associated with objects and files under their control.

5. Security. The kernel APSE design should be clear as to what extent it supports multilevel security. The security classification of an object or file represents a judgement of its sensitivity regardless of its accounting, protection, authentication, or storage partition and is therefore an independent partitioning. Security issues are not restricted to the usual notion of classification, but include any provisions for maintaining the integrity of the APSE or of systems developed using the APSE in responce to accidental or intentional damage.

F. History Preservation. The design should clarify how it satisfies the APSE requirement for recording and preserving information relating current and accessable noncurrent configurations. How does the proposed design manage the trade-off between the users desire to retain historical information that may be required later and the storage costs of many versions of objects that are never accessed? How does the kernel APSE determine or assist the user in determining which historical versions can be deleted?

G. Physical Input-Output Formats. The kernel APSE design must specify the physical representation used for the exchange of host system files, database objects, and Ada run-time data objects among host and target machines. Conversion of all data objects into a symbolic human readable form with all communications as text strings is simple and easy to define, but is expensive in the amount of data that must be transmitted and in the amount of processing required in both sending and receiving machines, and may be impractical for transfer of large volumes of data in distributed environments. A binary format close to that of machine representations would be less expensive in execution but is difficult to define independent of particular machine architectures. Physical input-output representations are needed for:

1. Text Files
2. Ada source programs
3. Ada intermediate language files
4. All data types in Ada programs
5. All objects of the APSE database

V. Other Host System Facilities

There are several other kernel functions and minimal APSE tools not discussed above.

A. Error Recovery Facilities. The kernel APSE must be designed and implemented to guarantee the integrity of the host system in the presence of hardware, software, and user errors.

1. Kernel APSE Protection. How does the kernel design prevent the user from accidentally or intentionally destroying, modifying, or accessing restricted portions of the kernel, APSE tools, and their data?

2. Error Diagnostics. How does the kernel APSE discover and isolate hardware and software errors within the host system? How does it identify and respond to inconsistancies in the APSE database?

3. Error Recovery. How does the kernel APSE respond to the discovery of hardware, software, or database errors? What does it do to recover the system? What does it report to the operator? to the user?

4. Fail Safe Execution. How does the kernel design ensure fail safe execution of the system? What provisions are made in the database updating operations to prevent destruction or partial loss of the database during system failures? How much of his current session can a user lose during a system failure? Is he given a clear and accurate report of the extent of any loss? Does the user get automatic restart at the point of failure?

5. Fail Soft Execution. Although not an APSE requirement, any provision in the kernel design for fail soft execution should be identified. Can the kernel reconfigure the host system in the presence of hardware failures (i.e., hardware subsystem loss) to limit the effect to restricting services or degradation in performance?

B. Performance Guarantees and Effiency. The utility and effectiveness of any APSE depends critically on its performance characteristics.

1. Operating System. The kernel APSE defines a virtual operating system which is machine independent. Implementation on top of an existing operating system gives access to existing tools of the system and permits the APSE to operate along with other nonAPSE systems, but may impose a performance penalty on the APSE user. How is the trade-off resolved in this kernel APSE design? How is the resulting performance degradation or access problem overcome in the design? How is machine dependence avoided?

2. Response Times. Does the kernel APSE provide any performance or response time guarantee for simple requests? What constitutes a simple request? Are there fixed levels of response which the user knows and understands (e.g., responses not made within one second will require 30 seconds)?

3. Levels of Service. Does the kernel APSE provide different classes of service? How are these determined? What guarantees are provided? What happens when the guarantee cannot be met?

4. Performance Degradation. What action is taken by the kernel when the demands for service exceed the available resources? What precautions are taken in the kernel design to avoid excessive consumption of resources by the kernel itself under overloaded conditions? Are performance limitations assigned uniformly, randomly, or to particular classes or users? How is this achieved in the design?

C. Ada Library and Assembly. The minimal APSE must provide a library manager for separate compilation of Ada programs and operations for assembling the code objects for a specified configuration into a load module for a given target machine.

D. Generic Help Facility. What standard conventions are proposed for subsystems and tools when reporting errors to users? What conventions are proposed for the user to query subsystems and tools about their status, what commands are available, and what syntax is proposed? What kernel or minimal APSE tools are provided to enforce or assist the tool developer in using these conventions?

E. Ada Compiler. Although the compiler must implement the full Ada language, there are a number of compiler design decisions that must be determined and reported to the APSE user and tool developer. Like most APSE features, these facilities should be defined as Ada packages.

1. Target Machines. What target instruction set architecture and target operating systems are supported? What is the minimum configuration supported for each target machine? What target perephirals are supported?

2. Compiler Pragmas. What pragmas are supported by the compiler? What are their parameter configurations, agrument options, and effect on the compilation process?

3. Foreign Language Interfaces. What, if any, languages does the compiler support for foreign code interfaces within Ada programs (i.e., the Ada INTERFACE pragma) and for what target machines?

F. Ada Interpreter. Does the command language interpreter include a full Ada interpreter? Is an Ada interpreter otherwise included among the minimal APSE tools? What, if anything, distinguishes the interpreter code from intermediate language for Ada program (see section II.B) and the semantically enchanced form in particular? What debug facilities are provided in conjunction with the Ada interpreter?

G. Network Interface. It is very likely that any APSE will be implemented in a local or broad based computer network. Those developed for the Department of Defense for example minimally should be accessable over the ARPA net.

1. Remote Users. How does the kernel APSE support remote (i.e., over a network) users of the APSE? Do remote users use the same interface conventions? Do they have access to the full set of APSE facilities? Do they enjoy the same quality of service and how is this guaranteed by the kernel APSE design?

2. Mail System. What mail system is provided among APSE users either locally of over a network? Is it compatible with existing mail systems of the network? Is it integrated with the APSE database? Is the mail system under program control so that it can be used as a basis for additional automation?

3. File Transfer. Does the kernel APSE provide access to the network file transfer protocol? If so how are transfers between the APSE and nonAPSE systems managed, prevented from violating the APSE protection mechanisms, and data formats converted?

4. Inter-APSE Cooperation. Is provision made for access to the APSE database at other network nodes, for cooperation among APSE°s of a network, or for load sharing among APSEs?

H. Other APSE Host Tools. What other host system tools are included in this minimal APSE design? What are their data structures and operations? How are they integrated with the other kernel and minimal APSE facilities? What tools or classes of tools beyond the scope of the minimal APSE have been considered in the kernel APSE design? Have sufficient handles been provided to allow later incorporation of these tools in individual APSEs?

VI. Target Machine Facilities

The concepts of separate host and target machines and cross compilation are fundamental to APSEs and impose requirements for target system tools, run-time support packages, and application libraries. The distinction between host and target systems in an APSE is somewhat blurred because the host machine is always one of the target machines for the Ada compiler of an APSE and host tools are written in Ada and thus can be compiled to any target machine with sufficient resources. The distinction is one of use, rather than location. As a general rule the APSE design must include the facilities of this section for each target machine (although in many cases, only recompilation will be required).

A. Target APSE tools. Certain tools are meaningful in the target environment.

1. Linker. A linker is required to complete those parts of the system assembly (see V.C.) which could not be done in the host system, to do address relocation, and to do any final parameterization or option selection as a function of the target machine configuration.

2. Loader. There must be tools capable of off-line or down-line loading of the final target code to the target system.

3. Dynamic Analysis Tools. Any facilities for snapshoting, breaking, tracing, monitoring, or timing execution must be provided as target system APSE tools. It is important that dynamic analysis tools report their findings to the user in Ada source program terms.

4. Postmortem Analyses. Static analysis of the target system at the point of a break or failure is often very useful in debugging or testing a system. Although the analysis itself can probably best be done on the host system, a target APSE tool is needed to extract the data from the target system and make it accessable to the analysis tools.

B. Ada Run-Time Package. Ada is a machine and operating system independent language. Thus standardization and the resulting portability of Ada software is achieved by including many of the traditional operating systems functions as language primitives and eliminating any need for explicit calls to the underlying (target) operating system. The price paid is that an Ada run-time package must be developed for each target machine (typically as an additional cost in developing the code generator portion of the compiler). Only those routines actually used by the target application, however, need by loaded and executed in the target machine.

1. Scheduler. Ada requires a first-in-first-out by priority scheduler to implement the tasking and rendezvous including entry calls and select.

2. Storage Manager. Ada requires a storage allocator and garbage collector for applications which use the full generality of access types (i.e., for uses in which static and stack storage disciplines are not sufficient and in which failure to dynamically recover inaccessable storage is too expensive).

3. Exception Propagation. The desire to avoid all execution overhead for exceptions which are not raised, imposes a requirement for a run-time routine to propagate exceptions when they are raised.

4. Real Time Clock. The run-time package must maintain a real time clock, make it accessable to the Ada target code in appropriate precisions and resolutions, and implement the delay operation.

5. Low Level Input-Output. The run-time package must implement the Ada low level input-output functions to provide a machine (but not device) independent Ada language level interface to those implementing device responders.

C. Standard Ada Library. The design and implementation of Ada compilers is simplified by providing definition facilities within the language and relegating many traditional language primitives to the standard library. The advantage is that the standard library functions need be implemented only once (in Ada) to become available to all compilers. The disadvantage is that a compiler is of little use without the standard library.

1. High Level Input-Output. The standard Ada high level input-output package must be implemented for each physical device to be supported.

2. System Package. The standard "system" package defines the machine dependent values of the compile-time program accessable target machine configuration description.

3. Standard Package. The "standard" package defines the predefined indentifiers for Boolean, integer, floating point, and character types.

4. Other Packages. Any other library packages to be provided with kernel or minimal APSE and Ada compiler should have a package specification and English language description in the design documentation. These might include a numeric package, definition of a variable length string type, or application data base package.

VII. Conclusions and Recommendations

This paper has been prepared in response to the immediate need for a list of items which can be checked when comparing and evaluating forthcoming designs from the three Air Force and one Army contracts for the design of the kernel and minimal APSE. There are an enormous number of issues that must be dealt with in the design of an integrated program support environment. We have attempted to catalogue as many of these as possible. There is, however, no reason to believe that this list is complete. Quite the contrary, the understanding of highly integrated software development and maintenance environments currently is so limited that a complete catalogue is probably not possible.

There are also limitations inherent in the catalogue approach. By concentrating on specific issues and features of the kernel and APSE designs, it detracts from the emphasis that rightfully should be given to other aspects of the design. These issues include:

- the important role that human factors should play throughout the kernel and APSE design,
- global issues related to ensuring simplicity and minimization of concepts as viewed by the APSE user,
- the importance of small granule tools and small granule interactions to achieve the synergy of APSE tools necessary to realize the potential inherent in the composition of tools,
- how modern Ada features, such as strong typing and information hiding, and "good" Ada style will alter the needs of an Ada based environment, and
- the critical requirement for common conventions among the various kernel APSE implementations.

Never-the-less we are hopeful that the catalogue will prove useful in the design and evaluation of the kernal APSE and that it will stimulate additional research and expreimentation on the many issues that are raised. In several cases we have barely been able to characterize the issues. It is unlikely that the designers will be able to address all issues with a high degree of confidence. Thus, the catalogue should be viewed as an aid in the design of the kernel APSE, but one that should be expanded and tuned as our understanding and experience in the design and use of integrated environments grow.

Finally, given the dearth of understanding of environmental issues and the limitations of the catalogue approach we recommend that it be supplemented by other tests during design and evaluation of the kernal and of APSEs. One promising approach, suggested in reference 3, is to examine tentative designs with respect to intended purposes and uses of the environment as seen from several different vantage points. For example, if the activities of documenting, debugging, and testing of Ada programs are not considered as integral to the APSE design, we might not be driven to necessary contemplating of the impact of documentation, debugging and testing on the kernel design decisions and vice versa. Will the kernel and APSE designs support the Software Lifecycle View, the Software Quality View, the Management Discipline View and the Maintenance and Enhancement View (see ref. 3) of an integrated environment? A combination of widespread public review and examination form a varity of view points coupled with analyses based on the catalogue should avoid the critical deficiencies of some existing environments and maximize the usefulness of the resulting kernel APSE and APSE systems.

REFERENCES

. *"Stoneman", Requirements for Ada Programming Support Environments*; John Buxton, et.al.; DARPA, (February 1980), 49 pp.

2. *Proceedings of Workshop on Environment, Certification and Control of DoD Common High Order Language*; T.A. Standish, ed.; University of California - Irvine, (June 1978).

3. *Initial Thoughts on the Pebbleman Process*, D.A. Fisher and T.A. Standish, IDA Paper P-1392, (June 1979), 73 pp.

4. *Ada Support system Study -- Requirements and Functions Specification*; Vic Stenning, et. al.; SDL and SSL, (March 1979), 34 pp.

5. *Ada Environment Workshop*; HOLWG; Harbor Island, San Diego, (November 1979).

6. *"Steelman", Department of Defense Requirements for High Order computer Programming Languages*; D.A. Fisher and P.R. Wetherall; HOLWG,(June 1978).

7. *Reference Manual for the Ada Programming Language -- Proposed Standard Document*; U.S. Department of Defense, (July 1980).